This assignment covers semantic analysis, including scoping and type systems. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 17, 2016 11:59 PM PDT. A \LaTeX{} template for writing your solutions is available on the course website.

1. (8 pts) Consider the following program fragment in Cool (using standard Cool type rules, scoping rules and general semantics):

```cool
1 class A {
2     f1(): Int {
3         let a: Int in {
4             a <- x;
5             a;
6         }
7     };
8     f2(): Int {
9         let a: Int in {
10             a <- 9;
11             let x: Int <- 5 in {
12                 a <- f1();
13             };
14             a;
15         }
16     };
17     x: Int <- 0;
18 };
19 class B inherits A {
20     f3(): Int {
21         let a: Int in {
22             a <- x;
23             let a: Int <- 6 in {
24                 a <- f2() + 3;
25             };
26             a;
27         }
28     };
29     f4(): Int {
30         let a: Int in {
31             a <- x;
32             let y: Int <- 7 in {
33                 let x: Int <- y in {
34                     y <- 8;
35                     a <- x;
36                 };
37             };
38             a;
39         }
40     };
41     f5(): Object {
42 }
```

```
42  let io: IO <- New IO, z: Int (* <- [Placeholder B] *) in {
43     let w: Int <- 2 in {
44         let z: Int <- w in {
45 (* [Placeholder A] *)
46             io.out_string("The secret will be: ");
47             io.out_int(z);
48         };
49         io.out_string("The secret is: ");
50         io.out_int(z); -- Statement 4
51     }
52  }
53 }
54 }
55 class Main {
56     main(): Object {
57         let o: B, io: IO <- New IO in {
58             o <- new B;
59             io.out_int(o.f2()); -- Statement 1
60             io.out_int(o.f3()); -- Statement 2
61             io.out_int(o.f4()); -- Statement 3
62             o.f5();
63         }
64     }
65 }

(a) (4 pts) For each of the three print statements in the method Main.main(), explain why it prints the value it does.

(b) (4 pts) Suppose you are told that you can only modify the code of the program fragment by replacing the comment including [Placeholder A] for any valid Cool expression you wish to write (your expression must match the Cool syntax for expr as given on page 16 of the Cool Manual). You are not allowed to edit the code anywhere else. Then, an adversary will take the code and replace only the comment including [Placeholder B] by a single valid Cool integer literal and run the program. Can you cause the line following [Placeholder A] to correctly predict the value printed in statement 4? If yes, provide the code fragment you should use to replace [Placeholder A]. If no, explain why. The resulting program after replacing both placeholders must be a valid Cool program.

2. (12 pts) Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[\text{Int}/y] \vdash y + y : \text{Int}$:

$$O[\text{Int}/y](y) = \text{Int} \quad O[\text{Int}/y](y) = \text{Int}$$

$$O[\text{Int}/y], M, C \vdash y : \text{Int} \quad O[\text{Int}/y], M, C \vdash y : \text{Int}$$

$$O[\text{Int}/y], M, C \vdash y + y : \text{Int}$$
Consider the following Cool program fragment:

```cool
class A {
    i: Int;
    j: Int;
    b: Bool;
    s: String;
    o: SELF_TYPE;
    foo(): SELF_TYPE { o };
    bar(): Int { 2 * i + j - i / j - 3 * j };
};
class B inherits A {
    p: SELF_TYPE;
    baz(a: Int, b: Int): Bool { a = b };
    test(c: Object): Object { (* [Placeholder C] *) };
};
```

Note that the environments $O$ and $M$ at the start of the method `test(...)` are as follows:

$$
O = \emptyset[Int/i][Int/j][Bool/b][String/s][SELF_TYPE_B/o][SELF_TYPE_B/p][Object/c][SELF_TYPE_B/self]
$$

$$
M = \emptyset[(SELF_TYPE)/(A, foo)][(Int)/(A, bar)][(Int, Int, Bool)/(B, baz)][(Object)/(B, test)]
$$

For each of the following expressions replacing [Placeholder C], provide the type derivation and final type of the expression, if it is well typed, otherwise explain why it isn’t. Assume Cool type rules (you may omit subtyping relationships from the rules when the type is the same, e.g. Bool $\leq$ Bool).

(a) (3 pts)

```cool
1    b <- p.baz(p.bar(),i)
```

(b) (3 pts)

```cool
1    p <- o.foo()
```

(c) (3 pts)

```cool
1    b <- baz(i+j,p.bar(i,o.foo()))
```

(d) (3 pts)

```cool
1    case c of
2        s: Int => s;
3        i: String => j;
4        b: Object => i;
5    esac
```
3. (8 pts) Consider the following Cool program fragment:

```cool
class A {
    i: Int <- 14;
    f(): Int { i };
}
class B inherits A {
    g(): Int { f() + f() };
}
class C inherits B {
    h(): Int {
        let x: A <- new C in {
            x.g() + x.f();
        }
    }
}
class Main {
    main(): Int {
        let c: C <- new C in { c.h(); }
    }
}
```

(a) (4 pts) This code does not compile. Provide a complete but succinct explanation as to why that is the case. Please note that the error message of coolc does not count as an explanation, your answer must show that you understand the problem.

(b) (4 pts) Assume you are given a variant of Cool which is dynamically typed instead of statically typed. Would the behavior of the code above be safe, in the sense of not triggering a runtime type error, in such variant of Cool? Why or why not?

4. (16 pts) Consider the following extension to the Cool syntax as given on page 16 of the Cool Manual, which adds arrays to the language:

```cool
expr ::= new TYPE[ expr ]
     | expr[ expr ]
     | expr[ expr ] <- expr
```

This adds a new type T[] for every type T in Cool, including the basic classes. Note that the entire hierarchy of array types still has Object as its topmost supertype. An array object can be initialized with an expression similar to “my_array:T[] ← new T[n]”, where n is an Int indicating the size of the array. In the general case, any expression that evaluates to an Int can be used in place of n. Thereafter, elements in the array can be accessed as “my_array[i]” and modified using an expression like “my_array[i] ← value”.

(a) (4 pts) Provide new typing rules for Cool which handle the typing judgments for:

\[ O, M, C \vdash new T[ e_1 ], O, M, C \vdash e_1[ e_2 ] \text{ and } O, M, C \vdash e_1[ e_2 ] \leftarrow e_3. \]

Make sure your rules work with subtyping.

(b) (4 pts) Consider the following subtyping rule for arrays:
\[ T_1 \leq T_2 \]  
\[ T_1[] \leq T_2[] \]

This rule means that \( T_1[] \leq T_2[] \) whenever it is the case that \( T_1 \leq T_2 \), for any pair of types \( T_1 \) and \( T_2 \).

While plausible on first sight, the rule above is incorrect, in the sense that it doesn’t preserve Cool’s type safety guarantees. Provide an example of a Cool program (with arrays added) which would type check when adding the above rule to Cool’s existing type rules, yet lead to a type error at runtime.

(c) (4 pts) In the format of the subtyping rule given above, provide the least restrictive rule for the relationship between array types (i.e. under which conditions is it true that \( T_1[] \leq T' \) for a certain \( T' \) or \( T'' \leq T_1[] \) for a certain \( T'' \)) which preserves the soundness of the type system. The rule you introduce must not allow assignments between non-array types that violate the existing subtyping relations of Cool.

(d) (4 pts) Add another extension to the language for immutable arrays (denoted by the type \( T() \)). Analogous to questions 4a and 4c, for this extension, provide: the additional syntax constructs to be added to the listing of page 16 of the Cool manual, the typing rules for these constructs and the least restrictive subtyping relationship involving these tuple types. It is not necessary that this extension interact correctly with mutable arrays as defined above, but feel free to consider that situation.

5. (12 pts) Consider another extension to the Cool language. In this case, we wish to add a special type to Cool that can either be an Int or a special value that represents “no result”. This \texttt{MaybeInt} type will take two forms: \texttt{Some(n)} where \( n \) is an integer, and \texttt{Nothing}. The compiler will provide two methods: \texttt{createSomething(n:Int):MaybeInt} and \texttt{createNothing():MaybeInt} which are defined in the \texttt{Object} class and produce each of the values of the \texttt{MaybeInt} type.

An example of where this type would be useful is a function like:

```plaintext
divide(numerator: Int, denominator: Int): MaybeInt {
  if (denominator = 0) then
    createNothing()
  else
    createSomething(numerator / denominator)
  fi
}
```

Which provides an implementation of an integer division that will not need to throw an exception when faced with a denominator of 0, but will return \texttt{Nothing} instead.

To be able to use the value inside of a \texttt{MaybeInt}, we add a pattern matching statement \texttt{match} to our language. Similar to how a \texttt{switch} statement works in other languages, \texttt{match} will go to a branch depending on the form of \texttt{MaybeInt} passed to it at runtime, while also possibly introducing a new value into the scope. The value of a \texttt{match} expression is the value of the expression on the right side of the branch that was taken. Example:
let div_result : MaybeInt in {
    div_result : MaybeInt <- divide(i,j);
    match(div_result) {
        Some(n) => "The result was: ".concat(n.to_string()),
        Nothing => "Can’t divide by zero."
    }
};

The grammar rule for \texttt{match} is:

\[
\text{expr ::= match (expr)\{Some(n) => expr, Nothing => expr\}} \quad (2)
\]

(a) (4 pts) Write a type checking rule for the \texttt{match} expression, which preserves Cool’s type safety guarantees.

(b) (8 pts) Give the operational semantics of the \texttt{match} expression. Consider referring to the operational semantics for Let and If-True/False from the Cool manual. Assume that when looking up \texttt{MaybeInt} in the store, it will either return \texttt{Some(n)} or \texttt{Nothing}, so you may need to write two separate rules.